AGNITAS EMM/OpenEMM Extension Development Guide

Table of Contents

1 Advantages of an Extension Architecture

Almost all big popular software products use an architecture which permits its enhancement via extension code (plugins), because it makes the software in general more

- flexible: features can be added (and removed) according to ones own needs
- modular: a lean core and a set of extensions instead of one big monolithic code monster
- maintainable: components can be installed, updated and removed independently from each other

To become more specific, an extension architecture has many advantages for both developers and users.

Advantages for developers:

- an extension developer does not have to comprehend the whole code of the software but only needs to understand its extension interface, consisting of extension points and the extension API
- the software core does not get bloated with functionality (less complexity, less dependencies)
- extensions can be maintained independently from the core software and vice versa
- updates of the software core are easier to execute because the only dependency on existing extensions are the locations of extension points and the signatures of API classes and methods
- customer-specific features do not have to be integrated into the software core
- when secondary features have to be refactored/extended they can be sourced out to an extension, keeping the core lean

Advantages for users:

- development time for customer-specific features implemented as extensions will drop, leading to lower costs
- extensions can be deployed at any time, independently from release cycles of the software core
- users can develop their own extensions
- users can use extensions developed by third parties
- if the code of the whole extension system is open source (like with EMM/OpenEMM), it makes comprehension of the extension interface even easier because it provides maxmimum transparency

2 Components of the EMM/OpenEMM Extension Architecture

The EMM/OpenEMM extension architecture consist of five components

1. Extension API
2. Extension Points
3. Extension Manager
4. Extension Registry
5. Extension Repository

Technically, the extension architecture of EMM/OpenEMM is based on the fabulous open source framework JPF (Java Plugin Framework), written by Dmitry Olshansky. JPF was inspired by the plugin architecture of Eclipse 2.x (before OSGi was introduced) and we think that it is a pity that it is no longer maintained actively. To learn more about JPF please visit http://jpf.sourceforge.net for details.

We have chosen JPF instead of the defacto highend standard OSGi because we feel that JPF is less complex than OSGi and that it provides everything that we need - but not more.

2.1 The Extension API

EMM/OpenEMM is very powerful and comes with a rich set of features. Accordingly, EMM/OpenEMM offers lots of interfaces to access bean objects and DAO classes with CRUD methods. To prevent re-inventing the wheel again we strongly suggest that ou use the existing interfaces.

You will find the basic bean interfaces in package *org.agnitas.beans* and the the CMS bean interfaces in package *org.agnitas.cms.beans*. The basic DAO interfaces are located in package *org.agnitas.dao* and the CMS DAO interfaces in package *org.agnitas.cms.dao*. The DAO interfacess provide a Javadoc documentation, and the names of properties and signatures of methods of the bean interfaces should be self-explanatory.

We also provide a Javadoc documentation for the Struts action classes in case you want to re-use those or just need ideas how to implement certain features. In general, if you want to know more about the design and the structure of the OpenEMM code, you should download the OpenEMM code design guide, available at https://sourceforge.net/projects/openemm/files/OpenEMM%20development/.

Please feel free to dig into the wealth of available public properties and methods and use them as you like. Only if you come across an deprecated annotation like

```
@Deprecated // since release 2012, alternative: ...
```

you should no longer use the property/method but choose the provided alternative because the property/method will be removed in a later release (at the earlist two years later).

Please note: If you want to use classes of packages *com.agnitas.** feel free to do so but be aware that in this case your extension will only work with EMM, not with OpenEMM!

To maintain integrity and consistency of database content the code of an extension must never change the content of the database tables directly with SQL statements. The only way to use the database tables is using the Extension API or your own DAO classes.

2.2 Extension Points

Generally, an extension point enables the core to execute code of an extension. An extension point is a pre-defined hook in the software core or in a base extension (the contact), which defines an interface (the contract) that will be implemented by a class of an extension that was registered for this extension point before.

To register an extension for an extension point the manifest file of the extension must assign a class of the extension to this extension point. When an extension is installed (by the extension manager) the info of its manifest file are written to the extension registry which in turn is accessed by an extension point to retrieve all classes bound to it.

In EMM/OpenEMM all basic extension points are supplied by system extensions. See section 3.6 for details.

2.3 Extension Manager

The extension manager (GUI term: plugin manager) is a sub-system of the software core with functionality to manage the lifecycle of extensions. It is visible for the user of EMM/OpenEMM if permission token *pluginmanager.show* is set (see chapter 7 for details).

The lifecycle stages of an EMM/OpenEMM extension are

Uploaded -> Activated -> Deactivated -> Removed

Uploaded: The ZIP file of the extension is uploaded, its manifest file is parsed, compatibility and dependencies are checked, the manifest configuration content is stored in the extension registry, and the files of the extension are unzipped and copied to the appropriate directories of EMM/OpenEMM

Activated: The extension is registered with all extension points defined in its manifest file and, therefore, made active. If the extension depends on an other extension, which is already uploaded but not activated yet, this extension is activated as well.

Deactivated: The extension is un-registered from all extension points. Any other extension which the extension depends on, remains activated.

Removed: The extension info are removed from the extension registry and all installed files of the extension are removed from the directories of EMM/OpenEMM

The initial list view of the extension manager shows a list of all installed extensions. The detail view of a single extension shows info like its name, version, vendor and dependency information, taken from the extension registry. In the detail view a extension can be activated, deactivated and removed. Via tab *Upload* new extensions can be uploaded from the file system.

2.4 Extension Registry

The extension registry is the central storage where the extension manager registers (and unregisters) extensions. The extension registry holds the necessary configuration data from all installed extensions and their manifest files (like version info, extension points, entries for extension points, etc.).

2.5 Extension Repository

We plan to offer an extension folder in the OpenEMM project at SourceForge (https://sourceforge.net/projects/openemm/files/) where public extensions will be available for download.

3 Components of an EMM/OpenEMM Extension

A typical extension for EMM/OpenEMM consists of

- a manifest file *plugin.xml* in XML format, holding configuration data to store in the extension registry (like version info, extension points, entries for extension points, etc.)
- resource files like property files for messages, navigation, etc.
- Java classes with program code of an extension (usually service layer classes, DAO layer classes and third-party JARs)
- JSPs to implement the GUI of an extension (sometimes accompanied with files for Javascript, HTML, CSS and images)

The extension manager takes care to put the files from the ZIP file of an extension in the appropriate directories of EMM/OpenEMM (see section 3.5).

3.1 Components of a Manifest File

The major components of a manifest file are:

- unique extension ID, version, name and vendor of extension (the alphanumeric UID will be used for directory names by the extension manager and should be used as prefix for new database tables and fields)
- various attributes like extension name, description and file name of messages properties file (tag *attributes*)
- required extensions for this extension by their extension IDs (tag *requires*)
- path to resource files and class files in ZIP file of extension (tag *runtime*)
- definition of GUI, navigation or feature extension points for use by subsequent extensions (tag *extension-point*)
- definition of bindings of own extension classes to extension points of underlying extensions like system extensions (tag *extension*)

Below you can see the content of a very simple manifest file (taken from system extension *core*):

```
<?xml version="1.0" ?>
<!DOCTYPE plugin PUBLIC "-//JPF//Java Plug-in Manifest 1.0"
"http://jpf.sourceforge.net/plugin_1_0.dtd">

<plugin id="core" version="0.0.1" vendor="Agnitas AG">

    <attributes>
        <attribute id="plugin-name" value="EMM core system" />
        <attribute id="plugin-description" value="System plugin providing core
extension points" />
    </attributes>

    <requires>
        <import plugin-id="core_navigation" />
    </requires>

    <!-- "URL"-extension point -->
    <extension-point id="featurePlugin">
        <parameter-def id="class" type="string" />
    </extension-point>

    <!-- Extension points for outputs to JSP -->
```

```
    <extension-point id="footer">
        <parameter-def id="class" type="string" />
    </extension-point>

    <extension-point id="admin.view.pos2">
        <parameter-def id="class" type="string" />
    </extension-point>

</plugin>
```

The XML format of the manifest file is defined by the Document Type Descriptor (DTD) of the JPF framework. You can find the DTD with lots of comments here:
http://jpf.sourceforge.net/dtd.html

The ID of the extension is *core*, its version is 0.1.0 and developer is Agnitas. The extension ID must be unique (if in doubt please check with us).

The following attributes provide a name and description of the extension which are shown in the extension manager.

Tag *requires* specifiy any dependencies. In this case the manifest file for *core* specifies that it is dependent on a second extension named *core_navigation*.

Finally, the manifest file specifies three extension points which may be used by other extensions. The first extension point is a feature extension point where code of extensions can be registered that provides its functionality. The second and third extension points are GUI extension points and allows an extension to insert output into the footer area and respectively the admin view of EMM/OpenEMM.

3.2 Resource Files

Resource files of an extension provide messages properties for the GUI, and navigation properties supply values for the navigation sidebar, submenus and tabs menus.

3.2.1 Messages Properties

You should never hard code messages in the GUI of an application. To be able to offer messages in different languages, EMM/OpenEMM offers a JSP custom tag *agn:message* which is similiar to tag *bean:message*. Tag *agn:message* needs two attributes: *plugin* and *key*. Attribute *plugin* defines the ID of the extension providing the messages properties file and attribute *key* is the message key. Example:

```
<agn:message key="key_name" plugin="dummy_feature" />
```

The name of the file providing all messages properties is defined in the manifest file of an extension within the attributes block like this:

```
<attribute id="i18n-bundle" value="messages-plugin" />
```

This means that the default messages file is *messages-plugin.properties*. And since the extension architecture of EMM/OpenEMM support I18n you can also provide localized messages files like *messages-plugin_de.properties* for a German language GUI or *messages-plugin_fr.properties* for French.

3.2.2 Navigation Properties

The name of the file providing the navigation properties is also defined in the manifest file of an extension. For each navigation extension point that has to be implemented a tag *extension* is needed in the manifest file. Tag *extension* defines the extension name and its extension point that has to be implemented, declares an ID for later reference and defines via parameter *navigation-bundle* the name of the file with the navigation properties for the extension point implementation. Example:

```
<extension plugin-id="core_navigation" point-id="tabs.feature" id="dummy-
navigation">
    <parameter id="navigation-bundle" value="dummyFeature" />
</extension>
```

In this case the (fictive) navigation extension point *tabs.feature* is extended by the navigation content of file *dummyFeature.properties*. The content of this file could look like this:

```
msg_1=dummy.Feature
href_1=/extension.do?feature=dummy-feature-point
token_1=dummy.feature
```

This content defines one new tab with message key *dummy.Feature* which is linked to the feature extension point implementation with ID *dummy-feature-point* and is only visible to the user if permission token *dummy.feature* is set for the user. (Please see chapter 7 to learn how to set permission tokens in the database.)

If you need more than one new menu item or tab, you have to name the properties of the second entry with msg_2, href_2 and token_2, and so on.

3.3 Java Classes

The functionality of an extension is usually implemented by the program code of one or more Java classes. The code of an extension is invoked by feature extension point *featurePlugin* (see section 4.3 for details) which is bound to a specific class of the extension. An extension class which is bound to the feature extension point is called gateway class in this guide because it provides the gateway to the feature code of an extension.

To implement feature extension point *featurePlugin* correctly the gateway class of an extension has to implement interface *EmmFeatureExtension* with its two callback methods

```
public void setup(PluginContext pluginContext, Extension extension,
ApplicationContextcontext)
```
and
```
public void invoke( PluginContext pluginContext, Extension extension,
ApplicationContext context)
```

Make sure that

- first argument is of type *org.agnitas.emm.extension.PluginContext*
- second argument is of type *org.java.plugin.registry.Extension*
- third argument is of type *org.springframework.context.ApplicationContext*

Object pluginContext provides three methods:

```
public HttpServletRequest getServletRequest();
```



```
public HttpServletResponse getServletResponse();
public void includeJspFragment(String relativeUrl)
```

The first two methods offers access to the HTTP request and HTTP response object and method *includeJspFragment* adds the JSP defined as input parameter to the current reponse so that it can access and display attributes set by an extension for the current request.

Object extension is the extension instance of the current extension taken from the extension registry. It offers various get methods to retrieve extension IDs, extension point IDs and other parameters. Please see the JPF API doc at http://jpf.sourceforge.net/api/index.html for details.

Object context is the Spring web application context which should be used to retrieve EMM/OpenEMM's DAO classes, bean classes, etc.

3.3.1 Setup()

Feature extension point *featurePlugin* first calls callback method *setup()* of the gateway class which has to assign message keys to attributes in the request scope. This step is separate from the invocation of the extension's code because the attributes are needed early in the rendering process of the GUI to generate the content for the navigation sidebar, the active submenu and the tabs menu. When the GUI is rendered message keys *agnTitleKey* and *agnSubtitleKey* are replaced by the language specifies values from file *messages-plugin.properties* via JSP *head-tag.jsp*.

List of navigation and menu attributes:

- sidemenu_active: active menu item in side menu
- sidemenu_sub_active: active submenu item in side menu
- agnTitleKey: message key for first text in browser title (title tag), defaults to "Extension" if not set
- agnSubtitleKey: message key for second text in browser title (title tag), defaults to content of attribute *name* in extension's manifest file if not set
- agnSubtitleValue: optional: parameter to insert in browser title (title tag)
- agnNavigationKey: name of navigation properties file for tabs menu items
- agnHighlightKey: name of highlighted menu item in tabs menu
- agnNavHrefAppend: optional: parameter appended to navigation links
- agnHelpKey optional: string for table doc_mapping_tbl which maps to a dedicated HTML page of the online user manual

3.3.2 Invoke()

After feature extension point *featurePlugin* has called method *setup()* of the gateway class, callback method *invoke()* is called to execute the extension's code which in the end generates the extension's output. See section 4.3 for more information.

3.4 Java Server Pages (JSPs)

JSPs are used to build the GUI of an extension. This may be a helpful example for a template of an extension JSP using a form:

```
<%@ taglib uri="/WEB-INF/agnitas-taglib.tld" prefix="agn" %>
```

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<c:url value="extension.do" var="form_url" />

<div class="grey_box_container">
    <div class="grey_box_top"></div>
    <div class="grey_box_content">
        <form action="${form_url}" method="post">
                <input type="hidden" name="feature" value="dummy-feature-
point" />
            <agn:message key="message" plugin="dummy.feature" />:
            <input type="text" name="value" />
            <input type="submit" name="<agn:message key="button"
plugin="dummy_feature" />" />
        </form>
    </div>
    <div class="grey_box_bottom"></div>
</div>
```

The first two lines include the tag libraries for custom tags like *agn:message* and JSTL tags like *c:url* so that the code for the tag can be located

JSTL tag *c:url* creates the full URL of the extension system's generic path *extension.do* and assigns it to parameter *form_url*. This URL is used by the action parameter of the form tag to define the target of the submit button.

The used DIV classes make sure that your layout fits neatly into the existing layout.

Hidden field *feature* make sure that request parameter *feature* is set to the ID of the feature extension point to be used.

The message for the key of the submit button is read from the properties file of the extension with ID *dummy_feature*.

If you want to use the powerful third-party tag *display* to show a list or a table in the JSP you should add line

```
<%@ taglib uri="http://displaytag.sf.net" prefix="display" %>
```

in the JSP header to specify its tag library so that the code for the *display* tag can be located. You may use the tag in the JSP body like this:

```
<display:table class="list_table" id="item" name="resultList"
excludedParams="*">
    <display:column titleKey="dummy.feature" class="dummy"
headerClass="dummy_head_name" property="feature" sortable="false" />
</display:table>
```

You can find the documentation for the display tag at http://www.displaytag.org. We strongly recommend its use because it eases the formatting of list and table output significantly.

3.5 Structure of an Extension ZIP File

The files of an extension have to be compressed into a single ZIP file with extension *.zip to produce a valid extension package. The pre-defined directory structure for an extension ZIP file looks like that:

plugin.xml	manifest file of extension
README.txt	optional readme file
/bin/classes	property files (*.properties)
/bin/classes/	Java class and interface files (*.class)
/bin/lib	Java libraries (*.jar)
/db	[install|remove]-[mysql|oracle].sql (up to 4 files containing SQL statements to install or remove database schema information or content, see chapter 6)
/jsp	JSP files (*.jsp)
/jsp/html	HTML and CSS files (*.html, *.css)
/jsp/images	images (*.png, *.jpg, etc.)
/jsp/js	JavaScript files (*.js)
/doc	documentation files (*.txt, *.pdf)
/src	source code files (*.java)

The two directories */doc* and */src* are not processed by the extension manager but may be included to provide information for the user like a manual, license information or the source code of the class files.

Best way to generate the ZIP file is to change to the directory level of manifest file *plugin.xml* and to create the ZIP file from there with

```
zip -r extension_name .
```

In this case the ZIP utility of Linux recursively compresses all files and subdirectories of the current directory into file *extension_name.zip*.

The extension manager unpacks the ZIP file and copies *plugin.xml* and all files from directories */bin*, */db* and */jsp* into a subdirectory named like the extension ID in directory *plugins* (called extension directory below). */bin* is copied to subdirectory *bin* , */db* is copied to subdirectory *_db* and */jsp* is copied to subdirectory *_jsp* of the extension directory.

The classloader of the extension system is able to load classes from the extension directory, which is outside of the webapps directory. But since it is not possible to execute the JSPs from outside of the webapps directory, the extension manager copies all files from directory */jsp* additionally to deployment location *webapps/openemm/plugins* into a subdirectory named like the extension ID.

At startup of EMM/OpenEMM the extension system copies all files from all */_jsp* subdirectories in directory *plugins* to the deployment location again. This makes sure that extensions survice an update of the core software, because otherwise JSP files would be lost since a core update would overwrite directory *webapps/openemm/plugins*.

3.5.1 Default Path Settings

These are the typical path settings used by OpenEMM:

DatabaseScriptsDirectory	/home/openemm/plugins/{pluginId}/_db
JspBackupDirectory	/home/openemm/plugins/{pluginId}/_jsp
JspBaseDirectory	webapps/plugins/
JspWorkingDirectory	webapps/plugins/{pluginId}/
PluginBaseDirectory	plugins.home in emm.properties (= /home/openemm/plugins)
PluginDirectory	/home/openemm/plugins/{pluginId}/
SystemPluginBaseDirectory	webapps/WEB-INF/system-plugins/
SystemPluginDirectory	webapps/WEB-INF/system-plugins/{pluginId}/

3.5.2 Path Settings in Manifest File

To make sure that the extension manager finds properties files, class files and JAR files in the ZIP file you have to specify their paths relative to directory */bin* within tag *runtime* of the manifest file like this:

```
<runtime>
    <library id="dummy-feature" path="classes/" type="code" />
    <library id="dummy-feature-lib" path="lib/dummy.jar" type="code"/>
</runtime>
```

The first entry defines that resource and class files will be found in subdirectory */bin/classes* and the second entry points to JAR file *dummy.jar* in subdirectory */bin/lib*. Basically, all entries within tag *runtime* are added to the classpath of the extension.

Please be aware that subdirectories in directory */bin* whose names start with underscore character "_" are not copied from the ZIP file for deployment to avoid name collisions, because the extension system uses directories starting with "_" for internal purposes.

3.6 System Extensions

EMM/OpenEMM comes with two system extensions which contain no own code but provide "root" extension points via their manifest files. These extension points can be be used by other extensions to hook into EMM/OpenEMM.

3.6.1 System Extension **core**

Manifest file *plugin.xml* of extension *core* (EMM: *emm_core*) supplies the global feature extension point *featurePlugin* which is the entry point for all extensions providing code enhancements for EMM/OpenEMM. See section 4.3 for more details.

Furthermore, *core* provides GUI extension points *footer* and *admin.view.pos2* as examples. These extension points may be used by an extension to insert its output in the GUI of EMM/OpenEMM. Right now there are only two GUI extension points in EMM/OpenEMM but we will add more GUI extension points in future releases. Section 4.1 demonstrates how you can add your own GUI extension points.

3.6.2 System Extension **core_navigation**

The manifest file of extension *core_navigation* (EMM: *emm_core_navigation*) supplies lots of navigation extension points to extend EMM/OpenEMM's navigation sidebar, its submenus and all tabs menus. The manifest file *plugin.xml* of *core_navigation* lists all available extension points. These extension points may be used to integrate new features seemlessly into the GUI of EMM/OpenEMM. See section 4.2 for more details.

Both extensions *core* and *core_navigation* must not be deactivated and removed because every extension will need at least one extension point to extend EMM/OpenEMM.

4 Extension Points in Detail

As mentioned before in section 3.6 EMM/OpenEMM offers three types of extension points:

- GUI extension points to insert output of an extension into an existing JSP
- Navigation extension points to add new menu items or tabs to the existing navigation
- Feature extension point to add new code which enhances the functionality

4.1 GUI Extension Points

A GUI extension point allows an extension to write output to the GUI of EMM/OpenEMM. System extension *core* provides the two GUI extentension points *footer* and *admin.view.pos2* by default. Extension point *footer* is located at the end of JSP *footer.jsp*:

```
<agn:JspExtensionPoint plugin="core" point="footer" />
```

and point *admin.view.pos2* is placed after the first content box of JSP *admin-view.jsp*:

```
<agn:JspExtensionPoint plugin="core" point="admin.view.pos2" />
```

If you want to set your own GUI extension points insert custom tag *agn:JspExtensionPoint* anywhere in the designated JSP. Tag *agn:JspExtensionPoint* uses attributes *plugin* and *point* which are required both. You can include the tag into an OpenEMM JSP like this:

```
<%@ taglib uri="/WEB-INF/agnitas-taglib.tld" prefix="agn" %>

<agn:JspExtensionPoint plugin="core" point="point.name.position" />
```

The first line should be in the header of the JSP and specifies the tag library so that the code for tag *agn:JspExtensionPoint* can be located. The second line must be placed in the JSP where the result of an extension should be included and it defines the extension the extension point belongs to (usually *core*) and the name of the point so that it can be addressed by an other extension.

If a JSP is processed to render the GUI, tag *agn:JspExtensionPoint* calls method

```
public void invokeJspExtension(String pluginName, String extensionPointName,
PageContext pageContext)
```

of class *ExtensionSystemImpl*, which forwards the call to method

```
invoke(extension, pageContext)
```

of the gateway class registered for this extension point (via the manifest file of an extension). Method *invoke()* can write content to the extension point directly with

```
context.getOut().println("Your output here");
```

Opposed to a feature extension point a setup method is not called by tag *agn:JspExtensionPoint* since a GUI extension point is located within an existing page and there is no need to set up any navigation attributes.

A new GUI extension point for system extension *core* would be defined in its manifest file *plugin.xml* in directory *WEB-INF/system-plugins/core* like this:


```
<extension-point id="point.name.position">
    <parameter-def id="class" type="string" />
</extension-point>
```

An extension implementing this extension would define the binding to its gateway class like this:

```
<extension plugin-id="core" point-id="point.name.position" id="dummy">
    <parameter id="class" value="com.domain.emm.plugin.dummy.DummyExtension" />
</extension>
```

As an example please have a look at the implementation of extension points *footer* and *admin.view.pos2* of system extension *core* by extensions *hello_world* and *admin_info* (see sections 5.1 and 5.2 for details).

After you have inserted a GUI extension point in a JSP file and added its definition to the manifest file of system extension *core* you do not have to recompile anything. A restart of the application is sufficient to activate the new extension point.

If you create GUI extension points in OpenEMM or need GUI extension points to be created in EMM please contact the author (address at the end of this guide) with info where your GUI extension point is located so that we will make sure that the next release of EMM/OpenEMM provides this extension point out of the box. (Since it is not possible for us to anticipate all future extension scenarios we will add new GUI extension points on demand.)

Extension point *admin.view.pos2* demonstrates nicely the naming conventions for GUI extension points. The first part of the extension point name consists of the name of the JSP file where the extension point is inserted (in this case *admin-view.jsp*), but the hyphen is replaced by a dot and the file extension is omitted. The suffix of the name describes the location of the extension point within the JSP file. The top position would be *pos1*, the second position after the first box would be *pos2*, the third position (after the second box) *pos3*, and so on.

If in later releases of EMM/OpenEMM the GUI is extended and there would be a need to insert extension points between *pos2* and *pos3*, for example, these points would carry suffixes like *pos23* or *pos27*.

4.2 Navigation Extension Points

EMM/OpenEMM's navigation sidebar, all submenus and all tabs menus provide extension points which are always located at the end of the list, i.e. at the lowest or the righter most position in the navigation sidebar, a submenu or a tabs menu. Each navigation extension point can be implemented by more than one extension so that it is possible to enhance a menu with more than one item.

As a rule of thumb you should follow these guidelines

New menu item in the navigation sidebar:

- an extension providing a new set of functionality should be integrated as a top-level menu item.
- example: a CMS module provides a new menu item “CMS” with submenus

New menu item in a submenu:

- an extension introducing a new feature should amend an existing menu with a new menu item

- example: a new wizard to export recipient's profile data could amend the recipient menu with a new menu item “wizard”

New tab:

- an extension implementing a new sub-feature should add a new tab to the page with the main feature
- example: a wildcard search for the blacklist should add a new tab “blacklist search” to the blacklist page (see section 5.3)

To integrate an extension in the navigation sidebar, a submenu or a tabs menu of EMM/OpenEMM an extension has to implement the matching navigation extension point. This is done by binding the file with the additional navigation properties of the extension to the extension point in the extension's manifest file like this:

```
<extension plugin-id="core_navigation" point-id="point_name" id="dummy">
    <parameter id="navigation-bundle" value="dummyFeature"/>
</extension>
```

This example extends (fictive) navigation extension point *point_name* of extension *core_navigation* with the content of navigation properties file *dummyFeature.properties*.

Parameter *href* of the properties file should be defined like this (see section 3.2.2 for details):

```
href_1=/extension.do?feature=dummy-feature-point
```

This ensures that extension point *dummy-feature-point* is invoked once the new menu item or tab gets clicked. Of course, this only leads to a result if the gateway class of the extension has been registered with feature extension point *featurePlugin* of *core* using ID *dummy-feature-point* (see next section for details).

Navigation extension points can be nested, that means the navigation properties file of an extension can contain extension points to extend the navigation of a first level extension by a second level extension. Example:

```
navigation.plugin=core_navigation
navigation.extensionpoint=navigation.main
```

This is in fact the extension point of the main menu of EMM/OpenEMM, located at the end of navigation properties file *sidemenu.properties*, and it is the first extension point defined in *plugin.xml* of system extension *core_navigation* for use by second level extensions:

```
<extension-point id="navigation.main">
    <parameter-def id="navigation-bundle" type="string" />
</extension-point>
```

4.3 Feature Extension Point

EMM/OpenEMM offers one global feature extension point named *featurePlugin*. This is the entry point for all extensions providing code enhancements for EMM/OpenEMM. All extensions that want to extend the functionality of EMM/OpenEMM have to implement this extension point. At first the gateway class of an extension has to be bound to the feature extension point like this:

```
<extension plugin-id="core" point-id="featurePlugin" id="dummy-feature-point">
```



```
    <parameter id="class"
value="com.domain.emm.plugin.dummyfeature.DummyFeatureExtension" />
</extension>
```

In this case gateway class *DummyFeatureExtension* has to implement interface *EMMFeatureExtension* with its two callback methods *setup()* and *invoke()*.

Alternatively your gateway class may extend class *AnnotatedDispatchingEmmFeatureExtension* which implements interface *EmmFeatureExtension* and provides some boilerplate code and offers a convenient way to call different methods via request parameter *method*. Annotation *@DispatchTarget* is used to map certain *method* parameters to certain methods in the gateway class. See demo extension *blacklist_wildcard_search* in section 5.3 for more information.

Of course, an extension can offer more than one feature and, therefore bind more than one gateway class to feature extension point *featurePlugin*.

5 Demo Extensions

The best way to really understand how an extension works is to have a look at a real world example.

Right now three demo extensions exist for EMM/OpenEMM: *hello_world*, *admin_info* and *blacklist_wildcard_search*. These extensions are quite simple because their purpose is to demonstrate how extensions work, using the three types of extension points and the API of EMM/OpenEMM. The extensions are available as ZIP files for download at SourceForge: https://sourceforge.net/projects/openemm/files/OpenEMM%20development

5.1 Extension *hello_world*

Extension *hello_world* extension is - aside from system extensions *core* and *core_navigation* - the most simple form of an extension. This extension uses GUI extension point *footer* to insert a support message in the footer of EMM/OpenEMM

Let's have a look at the files of *hello_world* to find out how the extension works in detail:

1. Manifest file **plugin.xml** (in the root directory of the extension ZIP file), providing
   - extension's name (“Hello World”)
   - name of messages properties file (“messages-plugin”)
   - dependency info (“core”, because GUI extension point *footer* is needed)
   - path to resource and class files in ZIP file's *bin* directory (“/classes”)
   - binding of gateway class *HelloWorldExtension* to GUI extension point *footer* of system extension *core*

2. Messages properties file **messages-plugin.properties** and local versions (in directory /bin/classes of the extension ZIP file)
   - provides the message, which is included in the footer of EMM/OpenEMM's GUI

3. Package *org.agnitas.emm.plugin.helloworld*, file **HelloWorldExtension.java** (in directory /src of the extension ZIP file)
   - implements method *invoke()*, called by the GUI extension point *footer*
   - inserts message in GUI footer via tag *JspExtensionPoint*
   - message is localized depending on the language of the current EMM/OpenEMM user

5.2 Extension *admin_info*

Extension *admin_info* is comparable to *hello_world* but it inserts an output which is a little bit more elaborate. *admin_info* uses GUI extension point *admin.view.pos2* of extension *core* to include a text box in JSP *admin-view.jsp*, which shows the details of an EMM/OpenEMM user account.

Extension *admin_info* consists of these files:

1. Manifest file **plugin.xml**
   - provides configuration and dependency information of extension
   - binds gateway class *AdminInfoExtension* to GUI extension point *admin.view.pos2*

2. Messages properties file **messages-plugin.properties** and local versions
   - provides the message, which is inserted in the page showing the details of an EMM/OpenEMM user account

3. Package *org.agnitas.emm.plugin.admininfo*, file **AdminInfoExtension.java** (in directory /src of ZIP file)
- implements method invoke(), called by GUI extension point *admin.view.pos2*
- inserts text box with message in JSP *admin-view.jsp* via tag *JspExtensionPoint*
- message is localized depending on the language of the current EMM/OpenEMM user

5.3 Extension *blacklist_wildcard_search*

Extension *blacklist_wildcard_search* does not only create output for the GUI of EMM/OpenEMM but it implements feature extension point *featurePlugin* to provide additional functionality in the form of feature “blacklist search”.

This feature consists of a new tab “Search blacklist” in submenu *Blacklist* of menu *Recipients*. When the new tab is selected the page shows a simple search box to enter the search expression. The search expression permits the use of wildcards “?” to indicate a single arbitrary character and “*” to represent an arbitrary sequence of characters.

After the search button is pushed the blacklist is searched and the result is shown on a new page as a list of email addresses.

The files of extension *blacklist_wildcard_search* are:

1. Manifest file **plugin.xml**
- provides configuration and dependency information of extension
- binds gateway class *BlacklistWildcardSearchFeature* to feature extension point *featurePlugin*
- implements navigation extension point *tabs.blacklist* with content of navigation properties file *blacklistSearchTabs*

2. Messages properties file **messages-plugin.properties** and local versions
- provides messages which are needed by callback method *setup()* of the gateway class and by the JSPs of the extension

3. Navigation properties file **blacklistSearchTabs.properties**
- provides the navigation properties for new tab “Search blacklist”

4. Package *org.agnitas.emm.plugin.blacklistwildcardsearch*, file **BlacklistWildcardSearchFeature.java**
- implements callback method *setup()* to set properties for the navigation sidebar, the active submenu and the tabs menu
- extends *AnnotatedDispatchingEmmFeatureExtension* so that request parameter *method* defines the name of the method which is called instead of default invocation method *invoke()*
- provides method u*nspecifiedTarget* which is called if request parameter *method* is not set; this method forwards to method *showForm*
- provides method *showForm* which simply forwards to JSP *blacklist-search-form.jsp* to request the search expression
- provides method *search* which is triggered by the search button of JSP *blacklist-search-form.jsp* and returns the search result via request attribute *resultList* to JSP *blacklist-search-result.jsp*

5. JSP **blacklist-search-form.jsp**
- shows form to enter search expression
- submit button calls generic extension URL with hidden request parameter *feature = blacklist-wildcard-search* (specifies the ID of the feature extension point

implementation to be called) and *method = search* (name of gateway class method to be invoked)

6. JSP **blacklist-search-result.jsp**
   - lists all search result with the help of third-party tag *display*

5.3.1 Program Flow of Extension

To better understand how extension *blacklist_wildcard_search* works a more detailed description of the program flow follows.

If tab “Search blacklist” is clicked link

```
/extension.do?feature=blacklist-wildcard-search
```

is called. In the end target *extension.do* is mapped to callback methods *setup()* and *invoke()* of gateway class *BlacklistWildcardSearchFeature* (see section 8.2 for mapping details).

Since *BlacklistWildcardSearchFeature* extends class *AnnotatedDispatchingEmmFeatureExtension* the value of request parameter *method* defines the method to be called. Annotation *@DispatchTarget* is used to map certain *method* parameters to certain methods in the gateway class:

```
@DispatchTarget(name="show-form")
```

If *method* has no value (which is the case initially) method *unspecifiedTarget* is called as fallback, which in return calls method *showForm*. *showForm* simply forwards to JSP *blacklist-search-form.jsp* to request the search expression.

```
pluginContext.includeJspFragment("blacklist-search-form.jsp");
```

blacklist-search-form.jsp shows a form so tha the user can enter the search expression. If the submit button of the form is pushed link

```
/extension.do
```

is called with with hidden request parameter *feature = blacklist-wildcard-search* and *method = search*:

```
<input type="hidden" name="feature" value="blacklist-wildcard-search" />
<input type="hidden" name="method" value="search" />
```

Request parameter *feature* specifies the ID of the implementation for feature extension point *featurePlugin* and parameter *method* defines (via @DispatchTarget) the name of gateway class method to be invoked.

The *search* method of gateway class *BlacklistWildcardSearchFeature* uses the search expression to search the blacklist and returns the search result via request attribute *resultList* to JSP *blacklist-search-result.jsp*:

```
pluginContext.includeJspFragment( "blacklist-search-result.jsp");
```

5.4 Extension *mailing_statistics_export*

Extension *mailing_statistics_export* is not a simple demo extension, but it provides a very useful feature. This extension enhances submenu *Comparision* of EMM/OpenEMM menu *Statistics* with a new *Export* tab which provides the functionality to export the values of mailing comparision statistics as CSV files for further analysis. You may filter the output written to the CSV file by several criterias:

- type of mailing
- start date and end date for send time
- type of statistics values to be exported

You can also define several settings for the CSV format like separator, text recognition character and charset to be used.

To analyse the code of this extension have a look at the source code of the Java classes in directory */src/org/agnitas/emm/plugin/mailingstatisticsexport*.

The structure of this extension is similiar to extension *blacklist_wildcard_search*, but in contrast to the blacklist extension, this extension spreads its code over three Java classes:

- Class **MailingStatisticsExportFeature** is the gateway class of the extension and implements EMM/OpenEMM's feature extension point *featurePlugin*. It provides core methods *selectProperties* to choose the export data and *doExport* to execute the final export and to create the export file.
- Class **MailingStatisticsExportWorker** implements a worker thread, so that the retrieval of the export data from the database (which may take a few minutes) is executed as a separate thread.
- Class **MailingStatisticsExportDataProvider** is the DAO class used to read the requested data from the database.

5.5 Using the EMM/OpenEMM API

Gateway class *BlacklistWildcardSearchFeature* demonstrates nicely how to use the API of EMM/OpenEMM. The header of the class shows which classes of OpenEMM are imported (EMM includes all OpenEMM classes):

```
import org.agnitas.beans.Admin;
import org.agnitas.beans.BlackListEntry;
import org.agnitas.dao.BlacklistDao;
import org.agnitas.util.AgnUtils;
import org.springframework.context.ApplicationContext;
```

Method *getAdmin()* of class *AgnUtils* is used to get the data of the current EMM/OpenEMM user, which is then stored in bean *Admin*:

```
Admin admin = AgnUtils.getAdmin(request);
```

The Spring web application context is used to retrieve the implementation of DAO class *BlacklistDao* :

```
BlacklistDao blacklistDao = (BlacklistDao)context.getBean("BlacklistDao");
```

Side note: Using the Spring context instead of creating the instance directly is always recommended because EMM might use a different class than OpenEMM. This is the case here because of differences in the database schema between EMM and OpenEMM, see chapter 6.)

Method *getBlacklistedRecipients* of *BlacklistDao* is used to access the blacklist of the company of the current user from the database and the returned entries are stored in a list of type *BlacklistEntry:*

```
List<BlackListEntry> list = (List<BlackListEntry>)
blacklistDao.getBlacklistedRecipients(admin.getCompanyID(), ...).getList();
```

6 Extending the EMM/OpenEMM Database

Some extensions will require the enhancement of the database schema of EMM/OpenEMM (see appendix A and B for details) or need to insert new data. If this is the case, please make sure that new tables use the extension ID as prefix in their name (like *dummy_contact_tbl*) and that new columns in existing tables are identified by this extension prefix as well to avoid overlap of schema enhancements by two different extensions.

The extension manager of EMM/OpenEMM is able to apply extension specific changes to the underlying database automatically. To do this you have to provide up to four files containing SQL statements to be processed in directory */db* of the extension ZIP file:

- `install-mysql.sql`: SQL statements to be executed if extension is installed and EMM/OpenEMM runs on MySQL
- `install-oracle.sql`: SQL statements to be executed if extension is installed and EMM/OpenEMM runs on Oracle
- `remove-mysl.sql`: SQL statements to be executed if extension is removed and EMM/OpenEMM runs on MySQL
- `remove-oracle.sql`: SQL statements to be executed if extension is removed and EMM/OpenEMM runs on Oracle

The extension manager checks which database is used, parses and validates the content of the corresponding SQL file and executes the individual SQL statements.

6.1 Table Names

Because EMM opposed to OpenEMM is a multi-tenancy application some tables are split per company:

OpenEMM	**EMM**
`cust_ban_tbl`	`cust{cid}_ban_tbl` (without field customer_id)
`customer_1_binding_tbl`	`customer_{cid}_binding_tbl`
`customer_1_tbl`	`customer_{cid}_tbl`
`mailtrack_tbl`	`mailtrack_{cid}_tbl`
`onepixel_log_tbl`	`onepixellog_{cid}_tbl`
`rdir_log_tbl`	`rdirlog_{cid}_tbl`

In case of EMM you have to insert the value of the company ID for expression {cid} to create the correct table name. For instance, if you want to read link click data for company_id 3, you have to query table rdirlog_3_tbl. (OpenEMM knows only company_id 1 and uses table rdir_log_tbl.)

if you want to work with blacklist table *cust_ban_tbl* be advised that for EMM this is a global table without column *company_id*. If you want to address the blacklist table of a certain company ID, please use table *cust{cid}_ban_tbl*. For company_id 3 this would be table *cust3_ban_tbl*.

Please also note that field *change_date* in OpenEMM tables is called *timestamp* in EMM tables!

If you want to write extensions that work with both OpenEMM and EMM out of the box the best way is to check the software with methods *isProjectEMM()* or *isProjectOpenEMM()* of class *AgnUtils*.

6.2 SQL Statement Validation

While the extension manager validates the SQL statements of a SQL file to be processed, validation is not perfect, of course, and you should take care to write valid and safe SQL statements.

Right now the validator checks if the names of new tables created by an extension use the extension ID + “_” as prefix and “_tbl” as suffix and it checks if new columns outside of extension's tables use the extension prefix as well.

Additionally, the validator checks that only those tables are dropped which use the extension ID + “_” as prefix and that columns in other tables are dropped only if they use the extension prefix as well. This is done to make sure, that no core tables, no tables from other extensions, no core columns and no columns from other extensions will be removed.

When writing SQL statements for a SQL file please make sure to follow these rules in order to prevent creating a mess in the database:

1. Do not alter, rename or drop core tables of EMM/OpenEMM (see appendix A for details).
2. Do not modify content of existing core tables of EMM/OpenEMM (see appendix A for details).
3. If you want to create new tables for an extension use the extension ID as prefix and suffix “_tbl” for all table names (like "dummy_name_tbl").
4. If you add new columns in tables which do not belong to your extension use the extension ID + “_” as prefix for the column name.
5. Do not write ALTER TABLE statements which mix ADD and DROP instructions, because they will be rejected by the validator of the extension manager.
6. While an ALTER TABLE statement to add new columns may add more than one new column, an ALTER TABLE statement to drop an existing column may only drop a single column, or else it will be rejected by the validator.
7. Make sure that no table name and no column name is longer than 30 characters in total - otherwise Oracle will not accept it.

7 Working with Permissions

Permission tokens are used in EMM/OpenEMM to show or hide menu items, features or sub-features in its GUI. In order to show an extension menu item or to show (or hide) a certain part of a feature the corresponding permission token has to be set in the database.

If you want a JSP to show the result of code if a certain permission token is set then use custom tag *agn:ShowByPermission* like this:

```
<agn:ShowByPermission token="dummy.feature">
insert your code here
</agn:ShowByPermission>
```

If you want a JSP to hide the result of code if a certain permission token is set use custom tag *agn:HideNavigation* like this:

```
<agn:HideByPermission token="dummy.feature">
insert your code here
</agn:HideByPermission>
```

To set the permission token in the EMM/OpenEMM database globally (4 ist the default admin group) use

```
INSERT INTO admin_group_permission_tbl (admin_group_id, security_token) VALUES
(4, 'dummy.feature');
```

And to set the permission token for an individual EMM/OpenEMM user use

```
INSERT INTO admin_permission_tbl (admin_id, security_token) VALUES
({id_of_admin}, 'dummy.feature');
```

Make sure to not use a permission token which is already taken by the core software of EMM/OpenEMM or by an other extension. To be on the safe side use the extension ID as prefix for your premission tokens.

Please note that a new permission token in the database requires a logout and a login to activate the change to the permission system.

8 Creating extensions for BIRT-based statistics (EMM only)

While OpenEMM simply uses HTML tables to render statistics, EMM uses the powerful BIRT framework to create more sophisticated statistics views like 3D pie charts. To create a new BIRT-based statistic you have to provide a report design file (**.rptdesign*) and at least one dataset class which provides the values to be displayed within the report design.

If you want to create an extension enhancing the BIRT-based statistics of EMM (“BIRT plugin”) you have to follow these guidelines:

8.1 BIRT Extension Points and Permission Token

To show the content of BIRT extensions in EMM its GUI was extended with additional sub-tab *Individual*. If permission *statistic.individual.show* is set for a certain EMM user, he/she will see this new sub-tab in tab *Statistics* of menu *Mailing*. Sub-tab *Individual* lists links to all BIRT extensions for which the user owns the required permissions.

To allow BIRT extensions to register their code for sub-tab *Individual* system extension *emm_core* provides special extension point *birt.statistics.mailing*. If a BIRT extension registers at this extension point the output of this extension will be shown at sub-tab *Individual*.

Extension point *birt.statistics.mailing* uses these three parameters:

titlekey message key for title of extension (its value must be listed in message properties file, see
section 3.2.1 for details)
reportdesign name of file providing the BIRT report design (*.rptdesign)
permissions list of permission token needed to display a link to the BIRT extension in sub-tab *Individual*
(see chapter 7 for details on permission token)

8.2 BIRT Extension ZIP File

For BIRT-related files use these directories:

/birt/rptdesign/ report design files (*.rptdesign)
/birt/scriptlib/ JAR file(s) with (dataset) classes for new report,
naming convention to prevent name collisions: birt-plugin-{pluginId}.jar

8.3 BIRT Configuration

To make sure that the extension manager knows where to put the BIRT-related files configuration file *emm.properties* of EMM has to define thes following properties:

birt.plugin.directory directory of BIRT server where the extension ZIP files will be copied to
birt.host.user user name to access BIRT server via SSH/scp
(i.e. BIRT server needs a user of this name without password)

On the BIRT server side file *emm-reporting.properties* needs this property:

plugins.home directory outside of webapps directory (to survive updates of BIRT software) where

extension ZIP files are copied to, identical to property *birt.plugin.directory* on EMM server side

8.4 Deployment of BIRT Extensions

The extension manager of EMM will install and un-install BIRT plugins like any other extension. However, since it is not possible to assign BIRT new Java classes at runtime you have to restart the BIRT server after installation and un-installation of a BIRT plugin to activate the change. This is what will happen exactly at restart time:

1. all rptdesign files installed by extensions will be removed
2. all JAR files installed by extensions will be removed
3. the directory defined in property *plugins.home* will be searched for extension ZIP files and all rptdesign files and JAR files will be unpacked and deployed

9 Inside the EMM/OpenEMM Extension Architecture

While you do not need to know the following technical details to write an extension for EMM/OpenEMM, it might be interesting for you to get a glimpse behind the curtain.

Since the extension architecture of EMM/OpenEMM is built on top of framework JPF (Java Plugin Framework) it might be a good idea to have a look at the extensive documentation of this framework as well: http://jpf.sourceforge.net

9.1 Extension System

Interface *ExtensionSystem* and its implementation class *ExtensionSystemImpl* wraps the main functionality of the plugin framework JPF and provides all basic methods:

- startup and shutdown of extension system
- initialize JPF framework (PluginManager, PluginRegistry, etc.)
- activate system extensions
- install, activate, deactivate and remove extensions
- invoke extensions implementing GUI and feature extension points
- various getter and setter for subsystems of extension system

Method *getPluginResource* of *ExtensionSystemImpl* uses classloaders of extensions to read resources like JARs and properties files because the extension classloaders know best where to find those resources and allow usage of different JAR file versions in different extensions.

9.2 Program Flow

When EMM/OpenEMM is launched Tomcat parses its deployment descriptor *web.xml* and starts listener *ExtensionSystemInitializationContextListener*. This listener initializes the extension system, activates all system extensions (*core* and *core_navigation* for now) and activates all extensions marked as active in EMM/OpenEMM database table *plugins_tbl*, column *activate_on_startup* (0 = do not activate, default: 1 = activate)

EMM/OpenEMM uses MVC framework Struts to separate model and view and templating framework Tiles to render its GUI. Normally, you could safely ignore Struts and Tiles but in order to fully understand the program flow a few information are necessary:

Whenever an extension is needed, URL path */extension.do* is called. Sometimes a request parameter feature is attached like

```
/extension.do?feature=dummy-feature
```

and sometimes the parameter is missing in the URL but transfered as hidden form field:

```
<input type="hidden" name="feature" value="dummy-feature" />
```

In Struts configuration file *struts-config.xml* */extension.do* is mapped to Struts class *ForwardAction* with parameter *extensionInTiles*, which means that the request is forwarded to Tiles definition *extensionInTiles*.

In Tiles configuration file *tiles-defs.xml* the generic extension definition *extensionInTiles* is mapped to path */extensionSetupServlet* for page setup (rendering of application frame) and to path */extensionServlet* for page body (rendering of application content).

In deployment descriptor *web.xml* path */extensionSetupServlet* is mapped to servlet *ExtensionSetupServlet* and path */extensionServlet* is mapped to servlet *ExtensionServlet*, both located in package *org.agnitas.emm.extension.web*.

Servlet *ExtensionSetupServlet* first retrieves the value of request parameter *feature*

```
String feature =
request.getParameter(ExtensionConstants.FEATURE_REQUEST_PARAMETER);
```

(`public static final String FEATURE_REQUEST_PARAMETER = "feature";`)

and after that calls method *invokeFeatureSetupExtension* of class *ExtensionSystemImpl* with parameter *feature* as extension ID:

```
extensionSystem.invokeFeatureSetupExtension(feature, applicationContext,
request, response);
```

Class *ExtensionSystemImpl* looks up the requested extension, retrieves its pluginContext object and calls callback method *setup()* of this extension:

```
featureExtension.setup(pluginContext, extension, context);
```

The task and the implementation of this method have been explained in section 3.3.1 in detail.

After servlet *ExtensionSetupServlet* has rendered the application frame servlet *ExtensionServlet* retrieves the value of request parameter *feature* as well and calls method *invokeFeatureExtension* of class *ExtensionSystemImpl*:

```
extensionSystem.invokeFeatureExtension(feature, applicationContext, request,
response);
```

Again, class *ExtensionSystemImpl* looks up the requested extension, retrieves its pluginContext object, but this time calls callback method *invoke()* of this extension with

```
featureExtension.invoke(pluginContext, extension, context);
```

to render the application content with the output of the extension (section 4.3 and 5.3 explain in detail how to implement method *invoke()*).

To include the output of an extension into a JSP, first the output has to be written to attributes of the current request. Second, class *PluginContextImpl* (which implements interface PluginContext) provides method *includeJspFragment* which may be used like this:

```
pluginContext.includeJspFragment( "dummy-feature-result.jsp");
```

To include a JSP into the current response so that it can read the request attributes, method *includeJspFragment* creates a RequestDispatcher as wrapper for the target JSP and uses its *include* method to insert the target JSP into the response object by adding it to the *out* buffer (of type *JSPWriter*):

```
public void includeJspFragment(String relativeUrl) throws IOException,
ServletException {
    request.getRequestDispatcher("plugins/" + this.pluginId + "/" +
relativeUrl).include(request, response);
}
```


This allows the target JSP to access and display request attributes set by the extension.

9.3 Main Components of the Extension Architecture

Libraries:

- JPF JAR files *jpf.jar*, *jpf-boot.jar*, *jpf-tools.jar* and *jxp.jar*

Listener:

- *ExtensionSystemInitializationContextListener*: Initializes the extension system and activates all system extensions at launch time of EMM/OpenEMM

Package org.agnitas.emm.extension(.impl)

- *AnnotatedDispatchingEmmFeatureExtension*: Implements EmmFeatureExtension, provides default method for *invoke()* which uses name of request parameter *method* to call final invocation method
- *DatabaseScriptExecutor*: Parses, validates and executes SQL script of extension to extend DB schema and/or DB content (entry point for parsing: class *SimpleSqlParser* in package org.agnitas.emm.extension.sqlparser)
- *EmmFeatureExtension*: Interface to be implemented by gateway class of extension, *setup()* sets navigation parameters, *invoke()* executes extension's program code
- *ExtensionManager*: Utility class with methodes to create and manage extension instances
- *ExtensionSystem(Impl)*: Main class to manage extension system, extensions and extension points
- *ExtensionSystemBuilder*: Initializes extension system at launch time of EMM/OpenEMM (used by listener *ExtensionSystemInitializationContextListener*)
- *ExtensionSystemConfiguration:* Configuration data for extension architecture (used by *ExtensionSystemBuilder*)
- *JspExtension*: Interface with method *invoke()* for extensions implementing GUI extension points
- *JspRestoreUtil:* Utility methods to copy JSP files (and accompanying files like images, JS files, etc.) from backup directory to deployment location
- *LocationTracker*: Manages locations (URLs) of extensions
- *PluginContext(Impl)*: Context of application like HTTP request and response, for use by extension
- *PluginFilenameFilter*: File name filter for locating extensions
- *PluginInstaller(Impl)*: Installs extensions from ZIP files
- *ResourceBundleManager*: Reads resource bundles for method *getPluginResourceBundle* of class *ExtensionSystemImpl*

Package org.agnitas.emm.extension.annotation

- *DispatchTarget:* Interface for annotation *@DispatchTarget* (convenience annotation, inspired by Struts DispatchAction)

Package org.agnitas.emm.extension.dao(.impl)

- *PluginDao(Impl)*: Interface/DAO class to access meta data of extensions from database table *plugins_tbl*

Package org.agnitas.emm.extension.exceptions

- various classes representing the custom exceptions of the extension architecture

Package org.agnitas.emm.extension.pluginmanager.[action|form]

- *PluginInstallerSelectAction*: Struts action class to select extension file upload, uses *plugininstaller-select.jsp*
- *PluginInstallerSelectForm*: Struts form bean for action class *PluginInstallerSelectAction*
- *PluginInstallerUploadAction*: Struts action class to initiate extension file upload (via *PluginInstallerImpl*)
- *PluginManagerAction*: Struts action class to manage extensions, uses *pluginmanager-list.jsp* for table of extensions and *pluginmanager-detail.jsp* for list of extension details

Package org.agnitas.emm.extension.sqlparser

- *CommandState*: Represents default parser mode “parsing a SQL statement”
- *MultiLineCommentState*: Represents parser mode “parsing a multi line comment” (indicated by “/*”)
- *ParserState*: Generic interface for all parser mode classes in this packages (**State*)
- *PossibleMultiLineCommentEndState*: Represents parse mode “possibly ending to parse a multi line comment” (indicated by “*”)
- *PossibleMultiLineCommentState*: Represents parse mode “possibly starting to parse a multi line comment” (indicated by “/”)
- *PossibleSingleLineCommentState*: Represents parse mode “possibly parsing a single line comment” (indicated by “-”)
- *QuotedStringState*: Represents parser mode “parsing a quoted string within a SQL statement” (indicated by quotation character “, ' or `)
- *SimpleSqlParser*: Implementation of SQL parser, entry point for this package with big constructor to set up parser configuration, parses SQL file, removes all comments and creates list of SQL statements to be executed
- *SingleLineCommentState*: Represents parser mode “parsing a single line comment” (indicated by “--”)
- *StatementBuffer*: Holds the intermediate result of the parsed SQL statement

Package org.agnitas.emm.extension.sqlparser.validator(.impl)

- *DatabaseScriptValidator*: Interface for SQL statement validation, implemented by class *SimpleDatabaseScriptValidator*
- *AlterTableAddColumnValidation*: Checks if SQL statements adding column(s) to an existing table use extension ID + “_” as prefix in column names
- *AlterTableDropColumnValidation*: Checks if SQL statements dropping a column dropped from an existing table drop only columns with extension ID + “_” as prefix in their names
- *BasicValidation*: Implementation of interface *StatementValidation* with basic validation code to be used by special-purpose validator classes
- *CreateTableValidation*: Checks if SQL statements creating new tables use extension ID + “_” as prefix for table names
- *DropTableValidation*: Checks if SQL statements dropping existing tables only drop tables with extension ID + “_” as prefix in their names
- *SimpleDatabaseScriptValidator*: Implements interface *DatabaseScriptValidator*, entry point for SQL statement validation, calls all validation classes for special-purpose validation, uses state machine pattern
- *StatementValidation*: Generic interface for validation classes

Package org.agnitas.emm.extension.taglib

- *ExtensionI18NTag*: Custom tag *agn:message* for localized messages in extensions
- *JspExtensionPointTag*: Custom tag *JspExtensionPoint* to set GUI extension points within JSPs

Package org.agnitas.emm.extension.util

- *ExtensionUtils*: Provides some auxiliary getter and setter methods
- *ExtensionConstants*: Defines some constants for the extension architecture
- *I18NFactory*: Factory to supply the appropriate resource bundle
- *I18NResourceBundle*: Special resource bundle class to be used by custom tag *agn:message*

Package org.agnitas.emm.extension.web

- *ExtensionServlet*: Servlet to call callback method *invoke()* of an extension (see section 8.2 for details)
- *ExtensionSetupServlet*: Servlet to call callback method *setup()* of an extension (see section 8.2 for details)

Package org.agnitas.taglib

- *ShowNavigationTag*:
  - code for custom tag *ShowNavigation*
  - used in JSP sidemenu-tiles.jsp to build navigation sidebar
  - used in tabsmenu.jsp to build tabs menus
  - modified for the extension architecture to process navigation data from navigation properties files of the core and extensions and to build new menu items or tabs with text and links, encapsulated by permissions
  - enhanced by method *prepareNavigationDataFromExtensionPoints* so that the class can not only read core navigation data from navigation properties files, but also read navigation data from properties files of extensions
  - uses method *getPluginResourceBundle* of class *ExtensionSystemImpl* to read navigation data from properties files of extensions
  - permits more than one level of extensions with nested navigation extension points

10 Documentation Requirements for Extensions

If you want your self-developed extension to be included in the public extension repository at SourceForge you should document it sufficiently. These are the minimum requirements for acceptance to the extension repository:

- Unique ID, name, version and description of extension (manifest file)
- required core version and extensions including their versions (manifest file)
- state of extension (pre-alpha, alpha, beta, release candidate, stable)
- basic description of functionality (incl. screenshots with English language GUI)
- name of original developer (single person, team or company, including email address)
- website of original developer
- software license (OpenEMM uses CPAL 1.0: http://opensource.org/licenses/cpal_1.0)

11 Todo List for Next Versions of Extension Architecture

- provide a template with documentation for a dummy extension
- introduce and explain handling of extension versions (manifest files support version handling, but right now versions of extensions are not used by demo extensions)
- include more GUI extension points so that existing JSPs can be enhanced individually with extension output (we need your input here where you need additional GUI extension points!)

If you have further questions regarding the extension architecture of EMM/OpenEMM or if you have suggestions how to improve this documentation please feel free to contact its author at maschoff@openemm.org.

12 Appendix A: Database Schema of OpenEMM

This appendix contains a description of the database tables of OpenEMM 2013 (“openemm”). To retrieve the datatypes of the columns of certain tables simply execute

```
$> mysql -u root -p
mysql> use openemm
mysql> desc {name};
```

and replace {name} with the name of the table you are interested in.

OpenEMM works with company_id 1 only, while EMM supports other values for company_id as well in order to offer its users multi-tenancy capabilities.

12.1 admin_group_permission_tbl

Table description:
Contains individual permissions for admin groups. The groups are defined in the admin_group_tbl.

Column name:	Description:
admin_group_id	ID of admin group (-> admin_group_tbl.admin_group_id)
security_token	Permissions of users in admin group

12.2 admin_group_tbl

Table description:
Contains admin groups. The permissions of the groups are registered in admin_group_permission_tbl. All admin_groups with company_id=1 are visible to all companies.

Column name:	Description:
admin_group_id	ID of admin group (auto-generated)
company_id	1
shortname	Name of the admin group
description	Description of the admin group

12.3 admin_permission_tbl

Table description:
Contains individual permissions of admins (users) to assign permission beyond group permissions.

Column name:	Description:
admin_id	ID of admin (-> admin_tbl.admin_id)
security_token	Individual permissions of admins

12.4 admin_tbl

Table description:
Contains info about admins (users). Permissions for admin_id=1 (“master user”) must be defined in table admin_permission_tbl.

Column name:	Description:
admin_id	ID of admin (auto-generated)
company_id	1
username	User name for login
fullname	User name shown in the frontend
admin_country	Home country of admin (us, de, etc.)
admin_lang	Language of admin (us, de, etc.)
admin_timezone	Timezone of admin
admin_lang_variant	Variation of timezone (like de_at)
layout_id	User-defined layout (-> emm_layout_tbl.layout_id)
pwd_hash	MD5 hash of password
pwd_change	Last change of password
creation_date	Creation date of this record
timestamp	Date of last change for this record
admin_group_id	ID of admin group(-> admin_group_tbl.admin_group_id)
preferred_list_size	Default number of entries in lists
default_import_profile_id	ID of default import profile (-> import_profile_tbl.id)

12.5 bounce_collect_tbl

Table description:
Temporary table used by softbounce processing script softbounce.py to minimize queries to (very big) bounce_tbl.

Column name:	Description:
mailtrack_id	ID of temporary record (auto-generated)
customer_id	ID of mail recipient (-> customer_1_tbl.customer_id)
mailing_id	ID of mailing (-> mailing_tbl.mailing_id)
company_id	1
status_id	Type of bounce (90 = softbounce)
change_date	Date of last change for this record

12.6 bounce_tbl

Table description:
Contains all soft and hard bounces without any filtering. Hard bounces are registered in the binding table as well. Soft bounces are processed daily and entered into another table for further processing (400 = other softbounce, 420 = problems with mailbox, 430 = problems with mailserver, 500 = irregular bounce, 510 = other hardbounce, 511 = unknown address, 512 = unknown domain) - very big table!

Column name:	Description:
bounce_id	ID of bounce (auto-generated)
customer_id	ID of mail recipient (-> customer_1_tbl.customer_id)
mailing_id	ID of mailing (-> mailing_tbl.mailing_id)
company_id	1
detail	Normalized DSN bounce code, 7 possible values (400, 420, 430, 500, 510, 511, 512)
dsn	DSN bounce code reported back by mailserver or pseudo code (499 for soft bounces, 599 for hard bounces)
change_date	Date of last change for this record

12.7 campaign_tbl

Table description:
Contains campaign (archive) information.

Column name:	Description:
campaign_id	ID of campaign (archive)
company_id	1
shortname	Name of campaign (archive)
description	Description of campaign (archive)

12.8 click_stat_colors_tbl

Table description:
Color values for percentage ranges in heatmap.

Column name:	Description:
id	ID of color (auto-generated)
company_id	1
color	Color value (hex code like FF0000)
range_start	Lower percentage value for range
range_ed	Upper percentage value for range

12.9 company_tbl

Table description:
Contains basic information on OpenEMM configuration.

Column name:	Description:
company_id	1
shortname	Name of company (default: “Agnitas Admin”)
description	Description of company (default: “Agnitas”)
status	Status of company (possible values: “active” and “inactive”, default: “active”)
creator_company_id	1
creation_date	Creation date of this record
xor_key	seed value for stronger link encryption , if you set/change value, links of sent mails will no longer work (default: “”)
timestamp	Date of last change for this record
notification_email	Email address receiving note for every data export (default: “”)
rdir_domain	FQDN for redirect link processing server (default: “http://localhost:8080”)
mailloop_domain	FQDN for mailloop processing server (default: “” which results in local FQDN)
mailtracking	1 (= enabled -> for every mailing every recipient's customer_id is written to mailtrack_tbl)
max_login_fails	maximum number of failed loging (default: 3)
login_block_time	number of seconds login is blocked after too many fails (default: 300)
uid_version	version of link encryption (default: NULL, do not change)

max_recipients	maximum number of recipients which is processed to display recipient list for GUI (default: 10,000)

12.10 component_tbl

Table description:
Contains content components of mailings (big table!).

Column name:	Description:
component_id	ID of component (auto-generated)
mailing_id	ID of mailing (-> mailing_tbl.mailing_id)
target_id	ID of target group expression (-> dyn_target_tbl.target_id)
company_id	1
compname	Name of component, reserved names start with agn (agnText, agnHtml, etc.)
description	Description of component
comptype	Type of component (0 = text, 1 = external image, 3 = attachment, 5 = embedded image)
mtype	MIME type of component (text/plain, text/html, image/gif, image/png, etc.)
binblock	Content of component of type 1, 3 and 5 as binary
emmblock	Content of component of type 0 as text
url_id	URL for image links (-> rdir_url_tbl.url_id)

12.11 config_tbl

Table description:
Contains generic configuration parameters (like link checker parameters).

Column name:	Description:
class	class or script for which the parameter is intended
classid	ID parameter (optional)
name	name of configuration parameter
value	value of configuration parameter

12.12 cust_ban_tbl

Table description:
Contains email adresses which must be excluded from mailings (blacklist).

Column name:	Description:
company_id	1
email	email address of recipient
creation_date	Creation date of this record

12.13 customer_1_binding_tbl

Table description:
Contains assignments of recipients of customer_1_tbl to mailing lists.

Column name:	Description:
customer_id	ID of recipient (-> customer_1_tbl.customer_id)
mailinglist_id	ID of mailing list(-> mailinglist_tbl.mailinglist_id)
user_status	Status of recipient's assignment to mailing list (1 = active, 2 = bounced, 3 = Opt-out by admin, 4 = Opt-out by user, 5 = Double-opt-in waiting for confirm , 6 = blacklisted, 7 = suspended)
user_type	Type of recipient (A = admin, T = test, W = normal (world))
user_remark	Comment for assignment (like "Opt-in-IP: aaa.bbb.ccc.ddd")
mediatype	Media type (0 = email)
exit_mailing_id	ID of mailing that generated hardbounce (for user_status = 2) (-> mailing_tbl.mailing_id)
creation_date	Creation date of this record
change_date	Date of last change for this record

12.14 customer_1_tbl

Table description:
Contains profile information of recipients. This table also holds all user-generated columns (see table customer_field_tbl). Assignments to mailing lists are saved in customer_1_binding_tbl.

Column name:	Description:
customer_id	ID of recipient
email	Email address
firstname	First name
lastname	Last name
title	Academic title or title of nobility
gender	Gender (0 = male, 1 = female, 2 = unknown)
mailtype	Mail type (0 = plain text, 1 = HTML, 2 = inline HTML)
datasource_id	Source for this record, only written on insert (-> datasource_description_tbl.datasource_id)
creation_date	Creation date of this record
change_date	Date of last change for this record

12.15 customer_field_tbl

Table description:
Contains information on user-generated columns (fields) in customer_1_tbl. The combination of company_id and col_name is the unique ID of the described column.

Column name:	Description:
admin_id	ID of user (-> admin_tbl.admin_id)
company_id	1
col_name	Name of the column (field) in customer_1_tbl
shortname	Name of column (field) in GUI
description	Description of column
default_value	Default value for column (field), if no other value is provided
mode_insert	Insert permission (0 = read+write, 1 = read only, 2 = hidden)
mode_edit	Edit permission (0 = read+write, 1 = read only, 2 = hidden)

12.16 datasource_description_tbl

Table description:
Contains source ID of recipient profiles. When importing a file a new datasource_id is generated and entered in table customer_1_tbl for each new recipient.

Column name:	Description:
datasource_id	ID of datasource
sourcegroup_id	ID of the sourcegroup (2, not used)
company_id	1
description	Description of datasource (file name)
creation_date	Creation date of this record
change_date	Date of last change for this record

12.17 date_tbl

Table description:
Formats accepted and used by the tag agnDATE

Column name:	Description:
type	ID, used for parameter type of tag agnDATE
format	Format of Date as accepted by Java class java.text.SimpleDateFormat (like "MM/dd/yyyy")

12.18 doc_mapping_tbl

Table description:
Maps pagekeys of JSPs to certain files in the online user manual for context sensitive online help .

Column name:	Description:
filename	Name of target file in directory of online user manual
pagekey	Name of key used in JSP to reference a certain help topic

12.19 dyn_content_tbl

Table description:
Contains content blocks for text modules.

Column name:	Description:
dyn_content_id	ID of content block (auto-generated)
dyn_name_id	ID of text module the content block belongs to (-> dyn_name_tbl.dyn_name_id)
target_id	ID of target group for content block (0 = no target group) (-> dyn_target_tbl.target_id)
company_id	1
dyn_content	Content block (text)
dyn_order	Position of content block within texte module, important for order of processing (1 = first, 2 = second, etc.)

12.20 dyn_name_tbl

Table description:
Contains text module names. Content for text modules is saved in table dyn_content_tbl.

Column name:	Description:
dyn_name_id	ID of text module (auto-generated)
mailing_id	ID of mailing the text module belongs to (-> mailing_tbl.mailing_id)
company_id	1
dyn_name	Name of text module, used for parameter name of tag agnDYN
deleted	Flag for deleted text module (1 = deleted)

12.21 dyn_target_tbl

Table description:
Contains code of target groups.

Column name:	Description:
target_id	ID of target group (auto-generated)
company_id	1
target_shortname	Name of target group
target_description	Description of target group
target_sql	SQL code used for the WHERE part of the SQL query to retrieve result set
target_representation	serialized instance of Java class TargetRepresentationImpl with target group definition
deleted	Flag for deleted target group (1 = deleted)
creation_date	Creation date of this record
change_date	Date of last change for this record

12.22 export_predef_tbl

Table description:
Contains templates for export profiles.

Column name:	Description:
id	ID of export template
mailinglist_id	ID of mailinglist (-> mailinglist_tbl.mailinglist_id)
target_id	ID of target group (-> dyn_target_tbl.target_id)
company_id	1
shortname	Name of export template
description	Description of export template
column_names	Semicolon-separated list of column names from customer_1_tbl for export file
mailinglists	Semicolon-separated list of mailinglist IDs whose user status should be added to export file
separator_char	Field separator (;,| or tab) used in export file
delimiter_char	Delimiter (no, " or ') for text in export file
charset	Character set used for export

user_status	Status of recipient's assignment to mailing list (1 = active, 2 = bounced, 3 = Opt-out by admin, 4 = Opt-out by user, 5 = Double-opt-in waiting for confirm , 6 = blacklisted, 7 = suspended)
user_type	Type of recipient (A = admin, T = test, W = normal (world))
deleted	flag for deleted export template (1 = deleted)
timestamp_start	Start date of change period
timestamp_end	End date of change period
creation_date_start	Start date of creation period
creation_date_end	End date of creation period
mailinglist_bind_start	Start date of mailinglist binding period
mailinglist_bind_end	End date of mailinglist binding period

12.23 import_column_mapping_tbl

Table description:
Contains mapping of columns in import files to database columns for import profiles.

Column name:	Description:
id	ID of (auto-generated)
profile_id	ID of import profile (-> import_profile_tbl.id)
file_column	Name of column in import file
db_column	name of column in customer_1_tbl
mandatory	Flag if column is required (0 = no, 1 = yes)
default_value	Default value for column if no value is provided in import file

12.24 import_gender_mapping_tbl

Table description:
Contains mapping of text gender descriptions to numeric gender values for import profiles.

Column name:	Description:
id	ID of (auto-generated)
profile_id	ID of import profile (-> import_profile_tbl.id)
int_gender	Numeric value of gender
string_gender	Alphanumeric value of gender

12.25 import_log_tbl

Table description:
Logs import results.

Column name:	Description:
log_id	ID of log record (auto-generated)
admin_id	ID of admin (user) (-> admin_tbl.admin_id)
datasource_id	ID of datasource (-> datasource_description_tbl.datasource_id)
company_id	1
profile	List of all parameters of selected import profile
imported_lines	Number of imported records
statistics	List of statistic information on import (# of existing records, new records, duplicates, errors, etc.)

creation_date	Creation date of this record

12.26 import_profile_tbl

Table description:
Contains user-defined import profiles.

Column name:	Description:
id	ID of import profile (auto-generated)
admin_id	ID of admin (user) (-> admin_tbl.admin_id)
company_id	1
shortname	Name of import profile
column_separator	Field separator (0 = ;, 1 = ,, 2 = |, 3 = tab)
text_delimeter	Text delimiter (0 = none, 1 = ", 2 = ')
file_charset	Character set used for import (0 = ISO8859-1, 1 = UTF-8, 2 = Chinese simplified)
date_format	Format for date fields (0 = dd.MM.yyyy HH:mm, 1 = dd.MM.yyyy etc.)
import_mode	Import mode (0 = add only, 1 = add+update, 2 = update only, 3 = opt-out, 4 = bounced, 5 = blacklist)
null_values_action	Handling of NULL values (0 = overwrite, 1 = ignore)
key_column	Name of key column used for duplicate checks
check_for_duplicates	Type of duplicate check (0 = import data only, 1 = complete, 2 = no check)
mail_type	Mail type (0 = plain text, 1 = HTML, 2 = inline HTML)
report_email	email address for import reports
update_all_duplicates	Flag for duplicate update mode (0 = only first, 1 = all)
ext_email_check	deprecated

12.27 login_track_tbl

Table description:
Contains all login tries to track failed logins, used for login block.

Column name:	Description:
login_track_id	ID of login try (auto-generated)
ip_address	IP address of login try
username	user name used for login try
login_status	10 = successful, 20 = failed, 40 = successful during blocking time
creation_date	Creation date of this record

12.28 maildrop_status_tbl

Table description:
When a normal mailing is sent or when a date-based or action-based mailing is activated, a new record is created for this table. The information is used by the backend to build and process the mailing.

Column name:	Description:
status_id	ID of mailing status (auto-generated)
mailing_id	ID of mailing (-> mailing_tbl.mailing_id)
company_id	1
gendate	Start time and date for generation of mailing

senddate	Start time and date for sending of mailing
blocksize	Number of recipients per XML data block
step	Time span in minutes between sending of consecutive XML data blocks
genstatus	Status of mailing generation (0 = generation is planned for a date in the future, 1 = generation can be started now, 2 = generation of mailing in progress 3 = generation of mailing is finished
genchange	Timestamp of last change to field genstatus
status_field	Type of mailing (A = admin, T = test, W = world (normal), E = event-based, R = rule-based (date based)

12.29 mailing_account_tbl

Table description:
While sending a mailing, for every sent block a record with the number, size and type of block is written.

Column name:	Description:
mailing_account_id	ID of mailing account entry (auto-generated)
mailing_id	ID of mailing (-> mailing_tbl.mailing.id)
maildrop_id	ID of maildrop status (-> maildrop_status_tbl.maildrop_id)
company_id	1
mailtype	Mail type of mailing (0 = plain text, 1 = HTML, 2 = inline HTML)
status_field	Mailing type (A = admin mailing, T = test mailing, w = World mailing, R = date-based mailing, E = event-based mail)
no_of_mailings	Number of sent emails
no_of_bytes	Size of all mails of this block in Byte
blocknr	Number of XML block causing this entry
change_date	Date of last change for this record

12.30 mailing_backend_log_tbl

Table description:
Contains information on how many mails of a mailing have been generated yet.

Column name:	Description:
mailing_id	ID of mailing (-> mailing_tbl.mailing_id)
status_id	ID of mailing status (-> maildrop_status_tbl.maildrop_status_id)
total_mails	Total number of mails of mailing
current_mails	Number of mails which have been generated
creation_date	Creation date of this record
change_date	Date of last change for this record

12.31 mailing_mt_tbl

Table description:
Contains the permitted media types for mailings and required configuration parameter for backend processing.

Column name:	Description:
mailing_id	ID of mailing (-> mailing_tbl.mailing_id)
mediatype	Media type (0 = email)
param	Configuration data for media type (email: from, subject, charset, linefeed, mailformat, reply and onepixel)

12.32 mailing_tbl

Table description:
Contains mailing and template information.

Column name:	Description:
mailing_id	ID of mailing (auto-generated)
mailinglist_id	ID of mailing list which the mailing is sent to (-> mailinglist_tbl.mailinglist.id)
campaign_id	ID of campaign (archive), the mailing belongs to, or 0 for no campaign (-> campaign_tbl.campaign.id)
mailtemplate_id	ID of template the mailing is based on or 0 (-> mailing_id)
is_template	0 = mailing, 1 = template (templates are saved in mailing_tbl as well)
company_id	1
shortname	Name of mailing
description	Description of mailing
mailing_type	0 = standard, 1 = date-based, 2 = event-based
creation_date	Creation date of this record
change_date	Date of last change for this record
target_expression	List with chosen target groups, separated with "|" and "&"
needs_target	1 = mailing needs to be assigned to target group
deleted	1 = mailing is flagged as deleted and not shown in GUI
archived	1 = mailing is used for online archive
cms_has_classic_content	flag for type of mailing content (0 = clean and right for CMS-based content, 1 = filled with classic content)
dynamic_template	defines if changes to template are copied to depending mailings (default: 0 = no)
openaction_id	ID of default adection for mail openings (-> rdir_action_tbl.action_id)
clickaction_id	ID of default action of link clicks (-> rdir_action_tbl.action_id)

12.33 mailinglist_tbl

Table description:
Contains mailinglist information.

Column name:	Description:
mailinglist_id	ID of mailinglist (auto-generated)
company_id	1
shortname	Name of mailinglist
description	Description of mailinglist

12.34 mailloop_tbl

Table description:
Contains mailloop (bounce filter) information.

Column name:	Description:
rid	ID of mailloop, also included in email address of mailloop
mailinglist_id	ID of mailinglist to subscribe to by email (subscribe_enable = 1) (-> mailinglist_tbl.mailinglist.id)
form_id	ID of form called on double-opt-in (-> userform_tbl.form_id)
shortname	Name of mailloop
description	Description mailloop
company_id	1
subscribe_enable	Subscribe by email (0 = disabled, 1 = enabled)
forward	Email address to forward mails to which were not filtered
forward_enable	Email forwarding (0 = disabled, 1 = enabled)
ar_sender	Sender address ofautoresponder
ar_subject	Subject line of autoresponder (if empty, original subject is preceded with prefix "Re: ")
ar_text	Text content of autoresponder
ar_html	HTML content of autoresponder (optional)
ar_enable	Status of auto responder (0 = disabled, 1 = enabled)
change_date	Date of last change for this record

12.35 mailtrack_tbl

Table description:
Contains a record for every recipient and every mail sent out to this recipient (big table!). Does not indicate whether the mail was finally received by the recipient.

Column name:	Description:
mailtrack_id	ID of mailtracking record (auto-generated)
customer_id	ID of recipient (-> customer_1_tbl.customer_id)
mailing_id	ID of mailing (-> mailing_tbl.mailing_id)
status_id	ID of mailing status (-> maildrop_status_tbl.status_id)
company_id	1
change_date	Date of last change for this record

12.36 onepixel_log_tbl

Table description:
Contains a record for every recipient who opened a mail.

Column name:	Description:
customer_id	ID of recipient (-> customer_1_tbl.customer_id)
mailing_id	ID of mailing (-> mailing_tbl.mailing_id)
company_id	1
open_count	Count of openings for recipient
ip_adr	IP address of recipient
change_date	Date of last change for this record

12.37 plugins_tbl

Table description:
Contains information on installed plugins.

Column name:	Description:
plugin_id	ID of plugin (auto-generated)
activate_on_startup	0 = do not activate, 1 = activate on startup (default)

12.38 rdir_action_tbl

Table description:
Contains user-defined actions.

Column name:	Description:
action_id	ID of action (auto-generated)
company_id	1
shortname	Name of action
description	Description of action
action_type	Scope of action (0 = links only, 1 = forms only, 9 = both)
operations	serialized instance of Java class from package org.agnitas.actions.ops with code of action

12.39 rdir_log_tbl

Table description:
Logs clicks on redirected links in sent mails.

Column name:	Description:
customer_id	ID of recipient (-> customer_1_tbl.customer_id)
mailing_id	ID of mailing (-> mailing_tbl.mailing_id)
url_id	ID of URL (-> rdir_url_tbl.url_id)
company_id	1
ip_adr	IP address of clicking recipient
change_date	Date of last change for this record

12.40 rdir_url_tbl

Table description:
Contains all measureable mailing links.

Column name:	Description:
url_id	ID of URL (auto-generated)
mailing_id	ID of mailing (-> mailing_tbl.mailing_id)
action_id	ID of associated action (-> rdir_action_tbl.action_id)
company_id	1
full_url	Genuine URL
shortname	Name of link
measure_type	Defines which mailtypes will be measured (0 = no, 1 = txt only, 2 = HTML only, 3 = both)

relevance	Relevance for statistics (0 = counted, 1 = not counted for gross click rate, 2 = not counted at all)

12.41 rulebased_sent_tbl

Table description:
Contains timestamps of rule-based (date-based) mailings to prevent sending more often than daily if senddate in maildrop_status_tbl is changed.

Column name:	Description:
mailing_id	ID of rule-based (date-based) mailing
lastsent	timestamp for last sending of mailing

12.42 softbounce_email_tbl

Table description:
Contains softbounce count for email addresses and is managed by softbounce processing script softbounce.py.

Column name:	Description:
email	email address of recipient
mailing_id	ID of mailing (-> mailing_tbl.mailing_id)
company_id	1
bnccnt	count of softbounces generated by recipient
creation_date	Creation date of this record
change_date	Date of last change for this record

12.43 tag_tbl

Table description:
Contains definition of tags for mailing content like agnEMAIL (simple) or agnDB (complex).

Column name:	Description:
tag_id	ID of tag (auto-generated)
company_id	1
tagname	Name of tag
description	Description of tag
selectvalue	Expression to be included in SQL statements which returns content for tag
type	Type of tag (SIMPLE = tag without parameters, COMPLEX = tag with parameters)
change_date	Date of last change for this record

12.44 timestamp_tbl

Table description:
Helper table to save timestamps from external scripts. Used by softbounce processing to restart at that point in time where previous run had finished.

Column name:	Description:

timestamp_id	ID of timestamp
description	purpose of timestamp
cur	current timestamp
prev	previous timestamp (optional)
temptemporary timestamp (optional)	

12.45 title_gender_tbl

Table description:
Contains gender-specific salutation textes for title types.

Column name:	Description:
gender	ID of genderID (0 = male, 1 = female, 2 = unknown)
title_id	ID of title (-> title_tbl.title_id)
title	text for salutation

12.46 title_tbl

Table description:
Contains title types for tag agnTITLE (salutations).

Column name:	Description:
title_id	ID of title (salutation) (auto-generated)
company_id	1
description	Description of salutation

12.47 userform_tbl

Table description:
Contains userforms.

Column name:	Description:
form_id	ID of user form (auto-generated)
company_id	1
formname	Name of form (should contain no special characters since it will be used in URLs)
description	Description of form
startaction_id	ID of action (-> rdir_action_tbl.action_id) executed at start of form processing
endaction_id	ID of action (-> rdir_action_tbl.action_id) executed at end of form processing if start action did not return error code
success_template	Code that is displayed in case of success (start action returned no error code), content is evaluated by Velocity before sent to browser
error_template	Code that is processed in case of error (start action returned error code), content is evaluated by Velocity before sent to browser
success_url	URL that is called in case of success (if success_use_url = 1)
success_use_url	flag to indicate if form or URL should be used in case of success (default: 0)
error_url	URL that is called in case of error (if error_use_url = 1)

error_use_url (default: 0)	flag to indicate if form or URL should be used in case of error

12.48 webservice_user_tbl

Table description:
Contains users of webservice API 2.0.

Column name:	Description:
username	Name of user
password	Password
company_id	1

12.49 ws_admin_tbl

Table description:
Contains users of webservice API 1.0.

Column name:	Description:
ws_admin_id	ID of webservice user (auto-generated)
username	Name of user
password	Password

13 Appendix B: Database Schema of CMS of OpenEMM

This appendix contains a description of the database tables of the CMS of OpenEMM 2013 (“openemm_cms”). To retrieve the datatypes of the columns of certain tables simply execute

```
$> mysql -u root -p
mysql> use openemm_cms
mysql> desc {name};
```

and replace {name} with the name of the table you are interested in.

13.1 cm_category_tbl

Table description:
Contains all categories of the CMS.

Column name:	Description:
id	ID of category
shortname	Name of category
description	Description of category
company_id	1

13.2 cm_content_module_tbl

Table description:
Contains content modules for CMS.

Column name:	Description:
id	ID of content module
category_id	ID of category (-> cm_category_tbl.id)
company_id	1
shortname	Name of content module
description	Description of content module
content	Structure of content module

13.3 cm_content_tbl

Table description:
Contains content of CMS placeholders in content modules.

Column name:	Description:
id	ID of content element
content_module_id	ID of content module (-> cm_content_module_tbl.id)
tag_name	Name of CMS placeholder tag
content	Content for CMS placeholder tag
tag_type	Type of CMS placeholder tag (0=TEXT, 1=LABEL, 2=IMAGE, 3=LINK)

13.4 cm_location_tbl

Table description:
Contains positions of content modules within CM templates and target group assignments.

Column name:	Description:
id	ID of location entry
mailing_id	ID of mailing (-> mailing_tbl.mailing_id)
cm_template_id	ID of CM template (-> cm_template_tbl.id)
content_module_id	ID of content module (-> cm_content_module_tbl.id)
target_group_id	ID of target group expression (-> dyn_target_tbl.target_id)
dyn_name	Name of agnDYN tag
dyn_order	Position of agnDYN tag within CM template (1=first, 2=second, etc.)

13.5 cm_mailing_bind_tbl

Table description:
Contains assignments of content modules to mailings.

Column name:	Description:
id	ID of assignment
content_module_id	ID of content module (-> cm_content_module_tbl.id)
mailing_id	ID of mailing (-> mailing_tbl.mailing_id)

13.6 cm_media_file_tbl

Table description:
Contains media files for CM templates, content modules and preview images (thumbnails).

Column name:	Description:
id	ID of media entry
cm_template_id	IF of CM template (-> cm_template_tbl.id)
content_module_id	ID of content module (-> cm_content_module_tbl.id)
cmtid	ID of CM type (-> cm_type_tbl.id)
company_id	1
media_name	Name of media file
content	Content of media file
media_type	Type of media file (0=file from dir "template-media" in template ZIP file, 2=preview image)
mime_type	MIME type of media file (image/gif, image/jpeg, image/png, etc.)

13.7 cm_template_mailing_bind_tbl

Table description:
Contains assignments of CM templates to mailings.

Column name:	Description:
id	ID of assignment
cm_template_id	ID of CM template (-> cm_template_tbl.id)
mailing_id	ID of mailing (-> mailing_tbl.mailing_id)

13.8 cm_template_tbl

Table description:

Contains CM templates for CMS.

Column name:	Description:
id	ID of CM template
company_id	1
shortname	Name of CM template
description	Description of CM template
content	Content of CM template

13.9 cm_text_version_tbl

Table description:
Contains default text versions for CM based mailings.

Column name:	Description:
id	ID of text entry
admin_id	ID of admin (user) (-> admin_tbl.admin_id)
text	text content

13.10 cm_type_tbl

Table description:
Contains CM types (content module templates) for CMS.

Column name:	Description:
id	ID of CM type
company_id	1
shortname	Name of CM type
description	Description of CM type
content	Structure of CM type
read_only	read/write flag (0=writeable, 1=read only)
is_public	public/private flag (0=visible to own company_id, 1=visible to all companies)